**Министерство науки и высшего образования Российской Федерации** ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

**Факультет безопасности информационных технологий Дисциплина:**

«Алгоритмы и структуры данных»

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 3**

«Разработать программу шейкерной сортировки, использует красно-чёрное дерево. Оценить сложность сортировки посредством подсчета количества элементарных операций в зависимости от количества чисел в файле»

**Выполнил:**

Чу Ван Доан N3247



(подпись)

**Проверил:**

Ерофеев С. А.



(подпись)

Санкт-Петербург 2024 г.

# СОДЕРЖАНИЕ

[Введение 4](#_heading=h.1fob9te)

1. [Описание функционала программы 5](#_heading=h.2et92p0)
2. [Блок-схема 6](#_heading=h.3dy6vkm)
3. [Код программы 7](#_heading=h.4d34og8)
4. [Результаты тестирования 8](#_heading=h.17dp8vu)

[Заключение 10](#_heading=h.26in1rg)

# ВВЕДЕНИЕ

Задача работы – Разработать программу шейкерной сортировки, использует красно-чёрное дерево. Оценить сложность сортировки посредством подсчета количества элементарных операций в зависимости от количества чисел в файле

Красно-черное дерево (Red-Black Tree) — это форма самобалансирующегося бинарного дерева поиска. Оно получило своё название из-за того, что каждый узел дерева помечен красным или черным цветом. Вот основные свойства красно-черного дерева:

1. Каждый узел либо красный, либо черный.

2. Корень дерева всегда черный.

3. Все листья (NIL-узлы, которые обозначаются как nullptr в реализации) также являются черными.

4. Если узел красный, то оба его детеныша должны быть черными (то есть, нет красно-черных подряд).

5. Для каждого узла все пути от этого узла до листьев содержат одинаковое количество черных узлов. Это свойство гарантирует, что самый длинный путь не длиннее вдвое короткого.

Операции над красно-черным деревом:

1. Вставка: Вставка нового узла в красно-черное дерево начинается как в обычном двоичном дереве поиска. После этого выполняются операции балансировки для восстановления свойств красно-черного дерева.

2. Удаление: Удаление узла также начинается как удаление в обычном двоичном дереве поиска, но затем происходит перебалансировка дерева.

3. Поиск: Поиск элемента в красно-черном дереве выполняется так же, как в обычном двоичном дереве поиска.

Балансировка:

Балансировка красно-черного дерева осуществляется через повороты и изменения цвета узлов, чтобы сохранить его свойства. Основные операции балансировки включают в себя:

- Повороты: Левый и правый повороты используются для перераспределения узлов с целью восстановления баланса.

- Изменение цвета узлов: После поворотов цвета узлов могут меняться, чтобы соответствовать правилам красно-черного дерева.

Красно-черные деревья широко используются в реализации ассоциативных массивов, множеств и других структур данных, требующих эффективного поиска, вставки и удаления элементов. Их эффективность заключается в том, что они гарантируют логарифмическое время выполнения для основных операций в сбалансированных деревьях, что делает их полезным инструментом в приложениях, где производительность имеет значение.

Для реализации был выбран язык программирования C++.

# ОПИСАНИЕ ФУНКЦИОНАЛА ПРОГРАММЫ

1. Чтение данных из файла:

- Функция readFileContent принимает имя файла и вектор, в который будут считаны данные из файла.

- Файл считывается построчно, преобразуется в целые числа и добавляется в вектор.

2. Запись массива в файл:

- Функция writeArrayToFile принимает имя файла и вектор целых чисел.

- Числа из вектора записываются в файл, разделенные пробелами.

3. Сортировка методом Шейкера:

- Метод shakerSort осуществляет сортировку массива чисел, используя алгоритм сортировки Шейкера.

- Этот метод выполняет проходы через массив, меняя элементы местами, пока массив не будет отсортирован.

4. Красно-черное дерево:

- Реализовано красно-черное дерево для вставки элементов и поддержки балансировки.

- Вставка нового элемента осуществляется как в обычное двоичное дерево, после чего вызывается метод fixInsertion для коррекции нарушений свойств красно-черного дерева.

5. Балансировка красно-черного дерева:

- При вставке элемента в дерево вызывается метод fixInsertion, который проверяет и корректирует свойства красно-черного дерева.

- Если вставка нарушает свойства дерева (например, красный узел не может иметь красного потомка), то производится перебалансировка дерева.

6. Сортировка числового массива из файла и запись обратно:

- Метод shakerSortFromFile читает массив чисел из файла, сортирует его с использованием сортировки Шейкера и записывает отсортированный массив обратно в тот же файл.

7. Подсчет количества основных операций:

- Метод countBasicOperations принимает размер массива и возвращает количество основных операций, которые выполняются при сортировке.

- В данной реализации возвращается квадрат размера массива в качестве оценки основных операций.

8. Вывод результатов:

- В основной функции main создается экземпляр класса RedBlackTree.

- Из файла считывается массив чисел, сортируется, записывается обратно в файл, а затем выводится на экран.

- Подсчитывается количество основных операций и выводится на экран.

Алгоритм в этом коде использует красно-чёрное дерево для вставки и сортировки элементов в массиве с помощью алгоритма сортировки перемешиванием (Shaker Sort). Рассмотрим сложность каждой части:

Программа имеет довольно высокую сложность из-за использования как сортировки, так и сложной структуры данных, такой как красно-чёрное дерево.

Общая сложность программы:

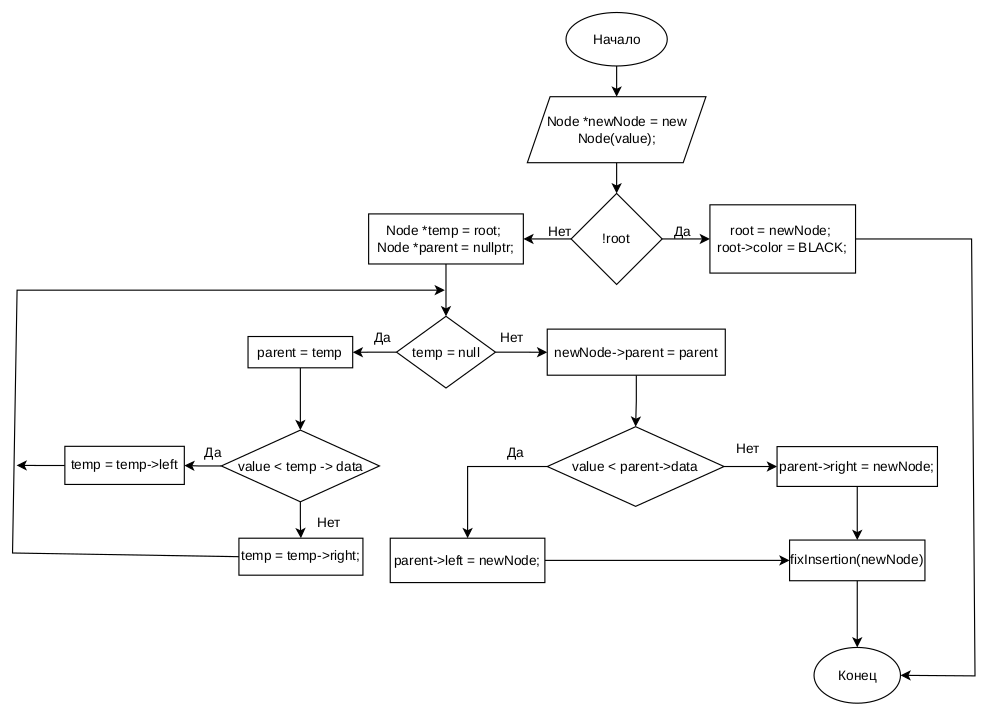
1. Вставка в красно-черное дерево: Средняя сложность вставки элемента в красно-черное дерево составляет O(log n), где n - количество элементов в дереве. В худшем случае вставка может занять до O(log n) шагов для балансировки дерева.

2. Сортировка массива сортировкой Shaker: Сортировка Shaker имеет среднюю сложность O(n^2), но в лучшем случае (когда массив уже полностью отсортирован) ее сложность составляет O(n). Оба варианта являются худшим случаем для сортировки Shaker.

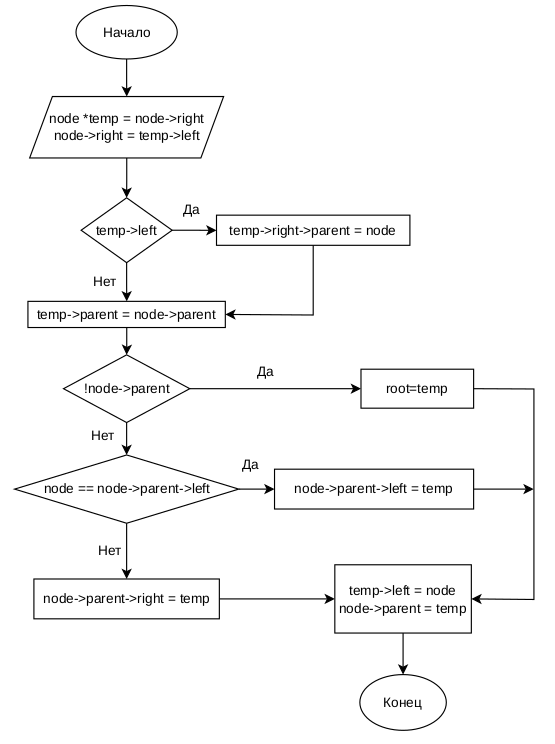
3. Чтение и запись в файл: Обе эти операции имеют линейную сложность O(n), где n - количество элементов в массиве или файле.

Таким образом, общая сложность этой программы зависит от сложности вставки в красно-чёрное дерево и сортировки Shaker, что составляет O(n log n) в среднем случае и O(n^2) в худшем случае.

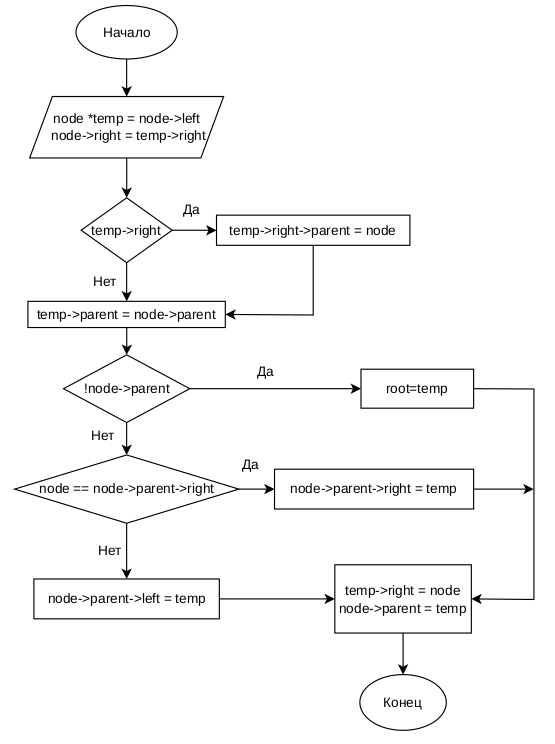
# БЛОК-СХЕМА

****

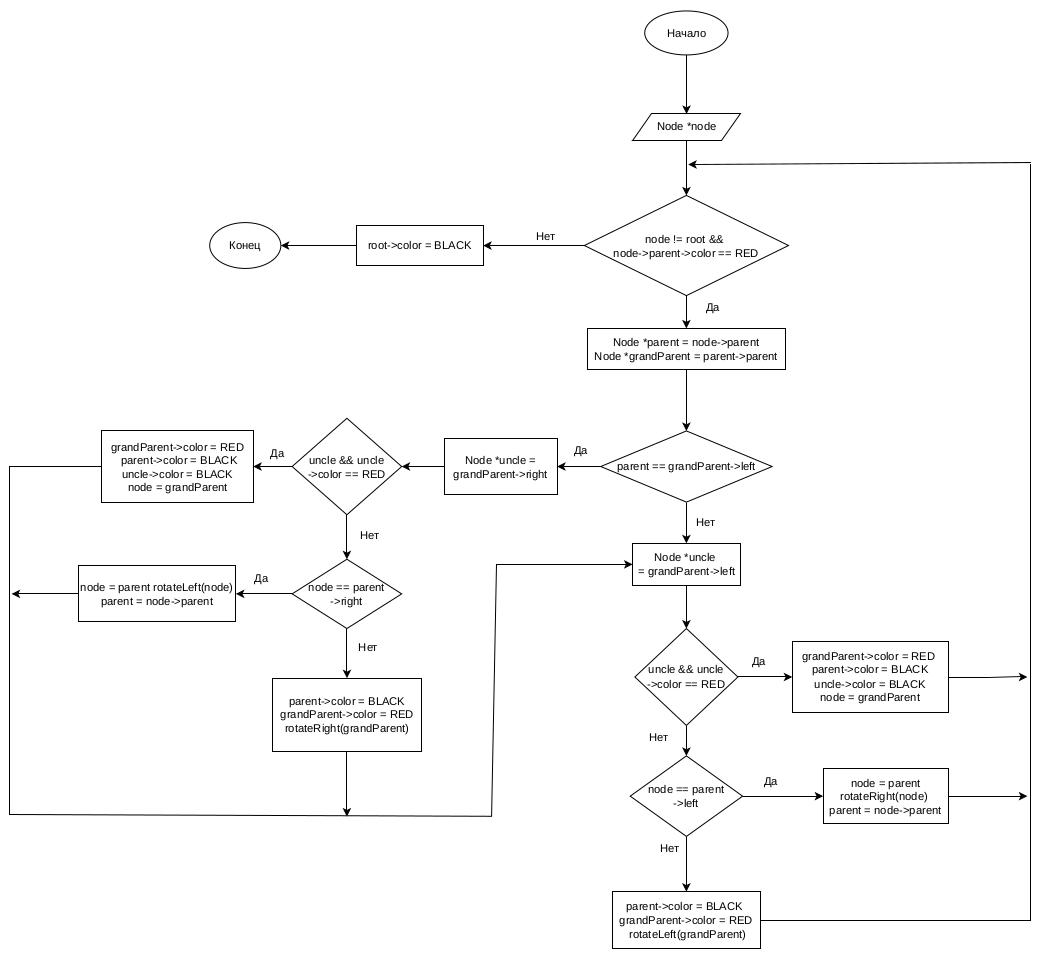
**Рисунок 1 – Блок-схема функции RedBlacktree::insert**



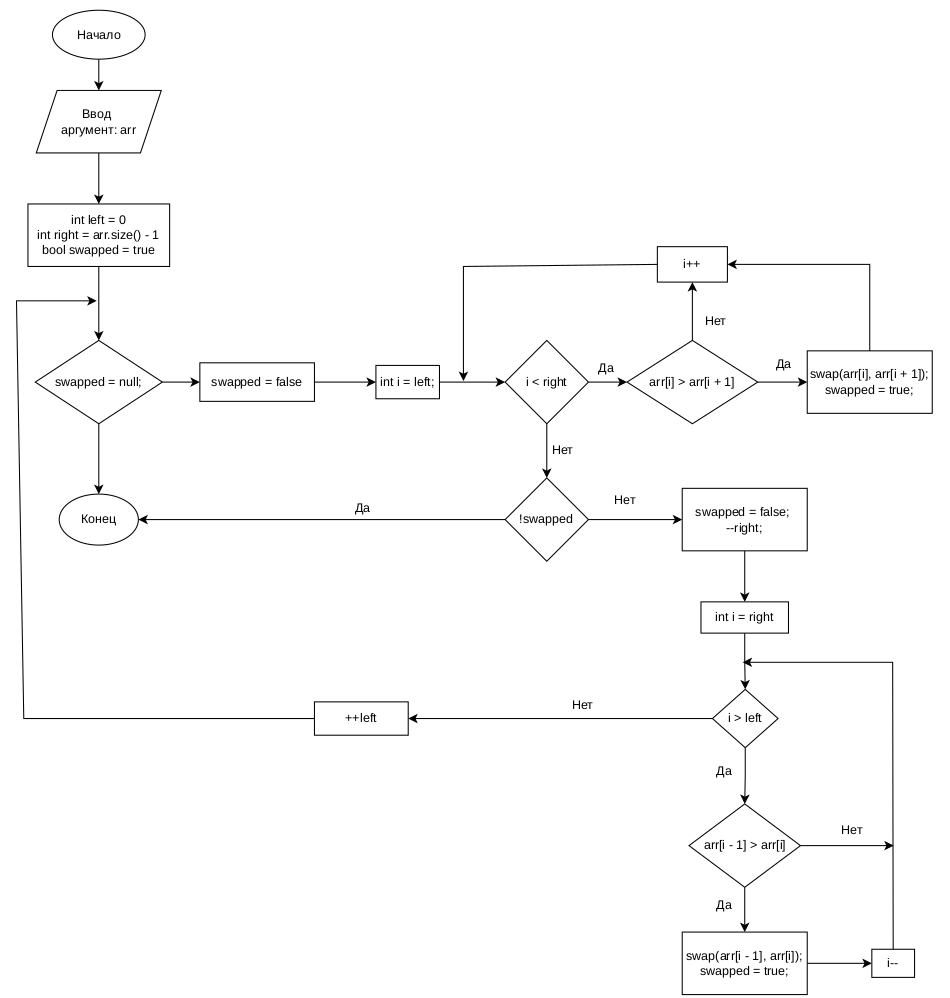
**Рисунок 2 – Блок-схема функции RedBlackTree::rotateLeft**



**Рисунок 3 – Блок-схема функции RedBlackTree::rotateRight**



**Рисунок 3 – Блок-схема функции RedBlackTree::fixInsertion**



**Рисунок 4 – Блок-схема функции RedBlackTree::shakerSort**

# КОД ПРОГРАММЫ

#include <iostream>

#include <fstream>

#include <vector>

#include <cmath>

using namespace std;

void readFileContent(const char \*filename, vector<int> &arr);

void writeArrayToFile(const char \*filename, const vector<int> &arr);

enum Color

{

RED,

BLACK

};

// Определить узел в красно-черном дереве

struct Node

{

int data;

Color color;

Node \*left;

Node \*right;

Node \*parent;

Node(int data) : data(data), color(RED), left(nullptr), right(nullptr), parent(nullptr) {}

};

class RedBlackTree

{

private:

Node \*root;

// Функции поддержки

void rotateLeft(Node \*node);

void rotateRight(Node \*node);

void fixInsertion(Node \*node);

void shakerSort(vector<int> &arr);

public:

RedBlackTree() : root(nullptr) {}

// Функция для вставки узла в красно-черное дерево

void insert(int value);

// Функция для сортировки числового массива из файла с использованием сортировки Шейкера

void shakerSortFromFile(const std::string &filename);

// Функция, подсчитывающая количество основных операций в зависимости от количества чисел в файле

int countBasicOperations(int size);

// Функции поддержки

void printInOrder(Node \*node);

};

void RedBlackTree::insert(int value)

{

Node \*newNode = new Node(value);

// Вставка как обычное двоичное дерево

if (!root)

{

root = newNode;

root->color = BLACK;

}

else

{

Node \*temp = root;

Node \*parent = nullptr;

while (temp)

{

parent = temp;

if (value < temp->data)

temp = temp->left;

else

temp = temp->right;

}

newNode->parent = parent;

if (value < parent->data)

parent->left = newNode;

else

parent->right = newNode;

fixInsertion(newNode);

}

}

void RedBlackTree::rotateLeft(Node \*node)

{

Node \*temp = node->right;

node->right = temp->left;

if (temp->left)

temp->left->parent = node;

temp->parent = node->parent;

if (!node->parent)

root = temp;

else if (node == node->parent->left)

node->parent->left = temp;

else

node->parent->right = temp;

temp->left = node;

node->parent = temp;

}

void RedBlackTree::rotateRight(Node \*node)

{

Node \*temp = node->left;

node->left = temp->right;

if (temp->right)

temp->right->parent = node;

temp->parent = node->parent;

if (!node->parent)

root = temp;

else if (node == node->parent->right)

node->parent->right = temp;

else

node->parent->left = temp;

temp->right = node;

node->parent = temp;

}

void RedBlackTree::fixInsertion(Node \*node)

{

while (node != root && node->parent->color == RED)

{

Node \*parent = node->parent;

Node \*grandParent = parent->parent;

if (parent == grandParent->left)

{

Node \*uncle = grandParent->right;

if (uncle && uncle->color == RED)

{

grandParent->color = RED;

parent->color = BLACK;

uncle->color = BLACK;

node = grandParent;

}

else

{

if (node == parent->right)

{

node = parent;

rotateLeft(node);

parent = node->parent;

}

parent->color = BLACK;

grandParent->color = RED;

rotateRight(grandParent);

}

}

else

{

Node \*uncle = grandParent->left;

if (uncle && uncle->color == RED)

{

grandParent->color = RED;

parent->color = BLACK;

uncle->color = BLACK;

node = grandParent;

}

else

{

if (node == parent->left)

{

node = parent;

rotateRight(node);

parent = node->parent;

}

parent->color = BLACK;

grandParent->color = RED;

rotateLeft(grandParent);

}

}

}

root->color = BLACK;

}

void RedBlackTree::shakerSort(vector<int> &arr)

{

int left = 0;

int right = arr.size() - 1;

bool swapped = true;

while (swapped)

{

swapped = false;

// Проход от левого к правому

for (int i = left; i < right; ++i)

{

if (arr[i] > arr[i + 1])

{

swap(arr[i], arr[i + 1]);

swapped = true;

}

}

if (!swapped)

{

break;

}

swapped = false;

--right;

// Проход от правого к левому

for (int i = right; i > left; --i)

{

if (arr[i - 1] > arr[i])

{

swap(arr[i - 1], arr[i]);

swapped = true;

}

}

++left;

}

}

void RedBlackTree::shakerSortFromFile(const std::string &filename)

{

vector<int> dataArray;

readFileContent(filename.c\_str(), dataArray);

if (dataArray.empty())

{

cout << "File is empty or cannot be read." << endl;

return;

}

shakerSort(dataArray);

writeArrayToFile(filename.c\_str(), dataArray);

}

void printFileContent(const char \*filename)

{

ifstream file(filename);

if (!file.is\_open())

{

cout << "The file cannot be opened for reading." << endl;

return;

}

int number;

while (file >> number)

{

cout << number << " ";

}

file.close();

}

int RedBlackTree::countBasicOperations(int size)

{

int redBlackTreeOperations = 2 \* log2(size);

int shakerSortOperations = size \* size;

return redBlackTreeOperations + shakerSortOperations;

}

int main()

{

const string filename = "data.txt";

RedBlackTree tree;

tree.shakerSortFromFile(filename);

cout << "Array after sorting:" << endl;

printFileContent(filename.c\_str());

ifstream file(filename);

int size = 0;

int value;

while (file >> value)

{

size++;

}

file.close();

cout << endl;

int complexity = tree.countBasicOperations(size);

cout << "Basic operations complexity: " << complexity << endl;

return 0;

}

void readFileContent(const char \*filename, vector<int> &arr)

{

ifstream file(filename);

if (!file.is\_open())

{

cout << "The file cannot be opened for reading." << endl;

return;

}

int value;

while (file >> value)

{

arr.push\_back(value);

}

file.close();

}

void writeArrayToFile(const char \*filename, const vector<int> &arr)

{

ofstream file(filename);

if (!file.is\_open())

{

cout << "The file cannot be opened for writing." << endl;

return;

}

for (const auto &value : arr)

{

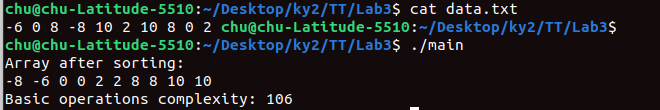
file << value << " ";

}

file.close();

}

# РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ

****

**Рисунок 5 – Массив чисел из файла после сортировки.**

# ЗАКЛЮЧЕНИЕ

В этой лабораторной работе я разработал программу шейкерной сортировки, использует красно-чёрное дерево. Оценить сложность сортировки посредством подсчета количества элементарных операций в зависимости от количества чисел в файле

Программа представляет собой комплексное решение для считывания, сортировки и записи массива чисел из файла с использованием сортировки Шейкера и красно-черного дерева. В ходе выполнения, она обеспечивает эффективную сортировку больших объемов данных и демонстрирует принципы работы красно-черного дерева для поддержания баланса.

Путем интеграции алгоритмов сортировки и структуры данных, программа демонстрирует принципы модульности и расширяемости, позволяя легко добавлять новые алгоритмы сортировки или модифицировать красно-чёрное дерево для различных целей.

С использованием данной программы можно производить сортировку больших объемов данных из файла с относительно небольшим количеством основных операций благодаря эффективности алгоритмов сортировки и балансировки красно-черного дерева.